Optimizing a 3D Multigrid Algorithm for the IA-64 Architecture

Markus Stürmer, Jan Treibig, Ulrich Rüde *

Abstract

Multigrid methods are known to be the numerically most efficient algorithms for solving large linear equation systems. Unfortunately a standard implementation runs with a, compared to the theoretical capabilities of modern processors, disappointing processor efficiency. While it is possible to achieve significant speedups with cache blocking techniques in 2D, 3D problems often don't benefit from optimizations to the same degree. The IA-64 architecture is a revolutionary new architecture to overcome the problems and limitations of the established dynamically scheduled, speculative execution processors. One of the key concepts is to put large parts of the complexity on the software side, hence the compilers. We present results for an highly optimized multigrid solver in 3D, showing the potential of the IA-64 architecture. Because the compiler is not yet capable to exploit the full potential of the architecture for our type of algorithm the computational intensive parts are implemented in assembly.

1 Introduction

Solving large linear equation systems, as they emerge in the field of numerical simulation from the discretization of partial differential equations, is a highly demanding task even for recent modern computer systems. Even when using multigrid methods, which offer nearly optimal complexity and are amongst the most effective solvers, algorithmic optimization is suggestive to exploit the available computational power. To demonstrate and study optimization techniques, a simple multigrid solver for Poisson's equation in 3D as outlined in [BHM00] was chosen as model problem. The cubic problem domain Ω is discretized by a regular grid with $2^n - 1$ unknowns in every dimension and only Dirichlet boundary conditions were allowed. A simple V-cycle with trilinear interpolation, full weighting restriction and a Red Black Gauss Seidel smoother with a seven point stencil is used further. The IA-64 architecture offers a much simpler execution logic than other established processors, capable only of in-order execution and lacking hardware prefetchers. On the other hand it provides a massively superscalar design with many execution units and an impressive fast and large cache hierarchy. Much of the complexity has been transfered to the compiler or assembler programmer that must create code much more diligently. Only by considering latencies when scheduling instructions and using the special concepts of the architecture good performance can be achieved. While this makes code generation more difficult, it also gives much more control over execution and enables better evaluation of

*Lehrstuhl für Systemsimulation, Institut für Informatik, Friedrich-Alexander Universität Erlangen-Nürnberg D-91058 Erlangen, Germany

the applied optimization techniques. For this transparency and massive hardware capabilities the Itanium 2 processor, actual implementation of this architecture, was chosen as test platform.

2 Optimizations

The following section presents a short summary of the applied optimization techniques: First a memory layout is outlined from which not only the smoother benefits, but also further optimizations of the V-cycle are enabled. A description of a blocking technique for the smoother follows and finally a modified V-cycle that needs fewer floating point operations and less memory throughput is outlined.

2.1 Memory Layout

The chosen memory layout described in [Stü05] splits the memory arrays for the unknowns and the right hand side into two memory arrays each according to the "color" they belong to in the smoothing step, where the definition of red and black points is extended to the boundary values analogously. This yields four arrays in total called *half arrays* for convenience from now on. As nearly all modern instruction sets support some kind of SIMD load operation (they are called Parallel Loads on the IA-64 architecture), the first unknowns in every line of the *half arrays* are aligned to 16 Byte boundaries and the lines are padded to equal length accordingly. Boundary values are then located before the first (effective at the end of the preceding line) or after the last unknown of the same color. While red and black points will reside in the same cache line when a naive memory layout is used, points of one color can be read and modified separately this way. Even for those cases, where memory transfer cannot be saved that way, this memory layout usually leads to less inter-cache transfer.

2.2 Blocking Technique for the Smoother

The smoother is the most expensive component of the multigrid solver, as well in terms of memory transfer and arithmetical operations [Wei01]. While there's few potential to reduce the number of floating point operations by reusing intermediate values, blocking techniques can perform one or more complete iterations per sweep (*temporal blocking*) and thereby reduce transfer to main memory significantly. The used blocking technique is based on blocks of two neighboring planes and is derived from [Stü05]. The calculation of i combined iterations starts with a special handling at the first planes: Therein the first red update is calculated in the first $2 \times i$ planes, followed by a black update in the first $2 \times i - 1$ planes and so on, until all i iterations have been completed in the first plane.
The blocked calculation itself begins with the first red update in the $2 \times i + 1$-th and $2 \times i + 2$-th plane and continues with the first black update in plane $2 \times i$ and $2 \times i + 1$ in a cascade like manner until the i-th black update in the second and third plane. This cascade is moved two planes further and so on until the other side is reached. A special border handling – analogous to the beginning – completes the computation. As long as all data needed for

two cascades can be held in caches, every operand has to be loaded into the cache hierarchy and, if modified, written back to main memory exactly once.
For greater plane sizes the introduction of another blocking level is required: The grid is split in x-z-plane into *super blocks* containing only a section of every plane, so that data can be held in caches long enough to use the above described blocking scheme effectively again. These *super blocks* must overlap, however, to fulfill data dependencies, and some operands must be transfered from and to main memory at least twice. The more iterations are combined in one sweep, the more lines this overlapping region will cover and the fewer lines a *super block* can contain either. The larger the caches and the shorter the lines are the bigger *super blocks* can be used on the other hand.

2.3 Optimization of the V-Cycle

With a faster smoother, the performance of the other V-cycle components becomes more and more important. For the given algorithm, however, a lot of computations can be saved without changing the result as shown in [Stü06].
The Gauss Seidel method works by setting the new value of a point to locally satisfy the underlying equation. This is equivalent to chosing the new value so that the point wise residual evaluates to zero. When using the Red Black Gauss Seidel variant, after an iteration the point wise residual on every black point will be zero within computational accuracy. If at least one Red Black Gauss Seidel step is used for pre-smoothing, calculation of residual and restriction can be limited to the red points. This not only saves nearly half of the floating point operations, but also lowers the amount of required fine grid data in conjunction with the suggested memory layout.
Further, for Red Black Gauss Seidel the new values of a red update depend only on black points and vice versa. Interpolation and correction can neglect red points as they would be overwritten afterwards, if at least one post-smoothing Red Black Gauss Seidel step is performed then.
Provided that at least one pre- and one post-smoothing Red Black Gauss Seidel step is done, an optimized, memory throughput and instruction reduced V-cycle can be used: After pre-smoothing, the residual for the red points is calculated and written into the memory locations of the corresponding unknowns, so that restriction has to take only this fine *half-array* into account. Later interpolation and correction are calculated together and only for the black unknowns on the fine grid. The first red update of the subsequent post-smoothing step will overwrite the no longer needed residuals with new approximations of the red unknowns.
While most optimized V-cycles combine calculation of the residual with restriction, it seems more appropriate here to calculate the red residuals blocked together with pre-smoothing just like an $i+1$-th red relaxation. After a V-cycle a residual norm can be computed on the finest grid level likewise, if desired. Thus calculation of residuals or a residual norm are nearly for free in terms of main memory transfer and primary limited by arithmetical in-cache performance. Only for larger planes the height of necessary *super blocks* must be reduced which induces some penalty.

3 Results

The above described optimizations were implemented in assembly language and tested on a HP zx6000 Workstation with two Itanium 2 processors running at 1.4 GHz, having 1.5 MB of Level 3 Cache each. The machine has 10 GB of double data rate SDRAM with a theoretical throughput of 6.4 GB per second. The main memory bandwidth is shared by the two processors.

Figure 1 shows the performance of the Red Black Gauss Seidel smoother. As reference the performance of a "straight forward" implemenation with an usual memory layout in Fortran is included. Its performance is compared with our fully optimized versions combining two or three iterations per sweep. To better use possibly free memory bandwidth a dual-threaded version was implemented, too (marked as parallel in the plot).

On this combination of processor and memory the smoother is theoretically memory bound for up to two iterations per sweep. The observed behavior confirmes this consideration as parallelization only enables a small speedup for two, but much more for three fused iterations, where the memory bus is able to feed another processor. For small grids parallelization overhead always induces a slowdown, however.

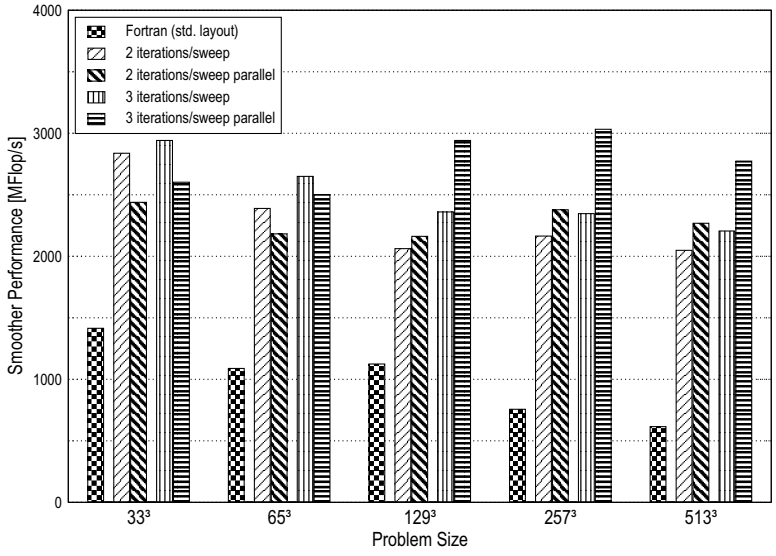

Figure 1: Results for Blocked 3D Red Black Gauss Seidel Smoother

Figure 2 visualizes the run time necessary to complete an overall V-cycle for a problem size of 513^3 (i. e. 511 unknowns in every dimension) including calculation of both a maximal and Euclidean norm. V-cycles with two, three and four pre- and post-smoothing iterations either were evaluated with separate calculation of the residuals and combined with the

smoother as well. Further, the performance with serial or parallel smoother is compared. If the calculation of the residual and its norms is melted with the smoother, efficiency of parallelization increases even more, as this is similar to "doing an additional half smoothing step".

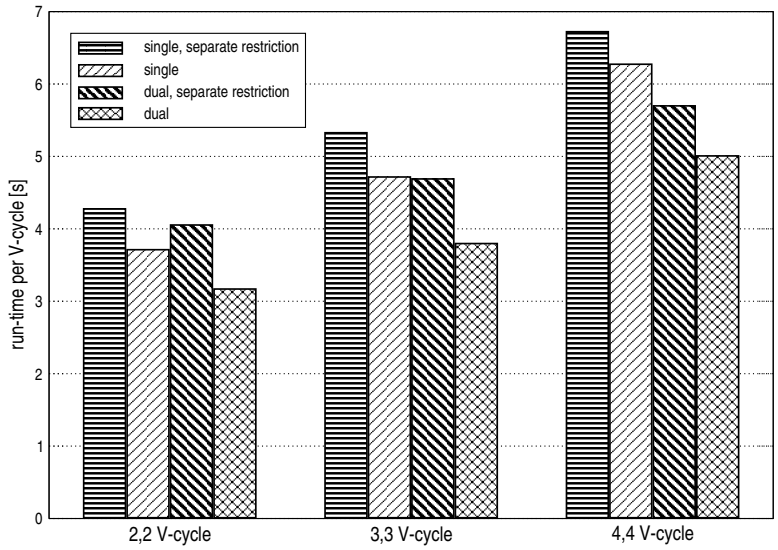

Figure 2: Runtime for one V-cycle, problem size 513^3

To give a frame of reference Table 1 shows a comparison in runtime for one V-cycle against other multigrid implementations. It has to be mentioned that HHG and ParExPDE are frameworks for production use and therefore offer a much higher flexibility and functionality compared to our pilot implementation. Still it can be seen how much performance is wasted by standard compiler generated code.

Code	time/V-cycle [s]	memory consumption [MB]
Optimized assembly 3,3 V-cycle	0.058	38
HHG (mixed Fortran/C++)	0.251	86
ParExPDE (C++)	0.88	1300

Table 1: Runtime of different multigrid implementations for 3D Poisson problem, 128^3 unknowns

4 Conclusion

We implemented a highly optimized geometric multigrid solving Poisson's equation in 3D and specifically tuned to the requirements of the Itanium 2 processor. The focus lies on showing how much improvement can be expected for this class of loop based iterative algorithms when optimizing for this specific architecture.

We have shown that it is possible to achieve speedups around factor 4-5 for the 3D Red Black Gauss Seidel smoother against a standard high level language implementation. Even more worth to mention is, that we succeeded in conserving this factor 4-5 to the full V-cycle. This paper shows that it is possible to exploit the full capabilities of a modern processor. Especially the Itanium 2 processor gives a lot of opportunities for low-level optimizations. It behaves in a transparent way giving a lot of control to the software side. One example for this is the very efficient software prefetching crucial for getting good performance on every modern processor.
While blocking and prefetching techniques can improve performance for nearly all Geometric Multigrid algorithms, possible algorithmic optimizations depend on which smoothers are applicable and which type of restriction is necessary.

References

[BHM00] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial*. SIAM, 2 edition, 2000. 1

[Stü05] M. Stürmer. Optimierung des Red-Black-Gauss-Seidel-Verfahrens auf ausgewählten x86-Prozessoren. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, August 2005. Studienarbeit. 2

[Stü06] M. Stürmer. Optimierung von Mehrgitteralgorithmen auf der IA-64 Rechnerarchitektur. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, May 2006. Diplomarbeit. 3

[Wei01] C. Weiß. *Data Locality Optimizations for Multigrid Methods on Structured Grids*. PhD thesis, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, Munich, Germany, December 2001. 2